Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE - SEMESTER–IV EXAMINATION – SUMMER 2025

Subject Code: 3140702 Date:08-05-2025

Subject Name: Operating System

Time: 10:30 AM TO 01:00 PM Total Marks:70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			Marks
Q.1	**(a)**	Explain in brief Program, Process and Thread.	**03**
	(b)	What is Process State? Explain different states of a process with various queues generated at each stage.	**04**
	(c)	Give the functions of Operating System and Explain following OS in detail. 1. Time Sharing 2. Real Time 3. Batch Operating System.	**07**
Q.2	**(a)**	Define following with respect to CPU Scheduling Algorithms: 1. Turnaround Time 2.Waiting Time 3. Throughput	**03**
	(b)	Compare Round Robin, FCFS and Shortest Job First Algorithms for CPU Scheduling.	**04**
	(c)	What is Semaphore? Give the implementation of Readers-Writers Problem using Semaphore.	**07**
		OR	
	(c)	What is Monitor? Give the implementation of Bounded Buffer Producer-Consumer Problem using Monitor.	**07**
Q.3	**(a)**	What is Shell and Kernel in Unix?	**03**
	(b)	List the necessary conditions that lead to deadlock.	**04**
	(c)	Explain Banker's Algorithm for Deadlock Avoidance with illustration..	**07**
		OR	
Q.3	**(a)**	Give the difference between Deadlock and Starvation.	**03**
	(b)	What is Process Control Block (PCB)?Explain various entries in PCB.	**04**
	(c)	Draw the block diagram for DMA. Write steps for DMA data transfer.	**07**
Q.4	**(a)**	Explain Internal and External Fragmentation in Memory Management.	**03**
	(b)	Explain the Linux/Unix Commands: cat, rmdir, sort, chmod.	**04**
	(c)	Explain FIFO, LRU and Optimal Page Replacement Algorithms with example.	**07**
		OR	
Q.4	**(a)**	Write a shell script to find a factorial of given number n.	**03**
	(b)	Compare Paging and Segmentation with reference to Memory Managament.	**04**
	(c)	Explain various Disk Arm Scheduling Algorithms for Disk Space Management.	**07**
Q.5	**(a)**	Explain difference between Security and Protection?	**03**
	(b)	What are the Allocation Methods of a Disk Space?	**04**
	(c)	Explain in Detail File and Directory Management of Unix/Linux Operating System in detail.	**07**
		OR	
Q.5	**(a)**	What is Access Control List?	**03**

(b) Explain the concept of virtual machines and pure virtualization. **04**

(c) What is RAID? Explain various levels of RAID and it importance. **07**

****

Operating System SUMMER 2025

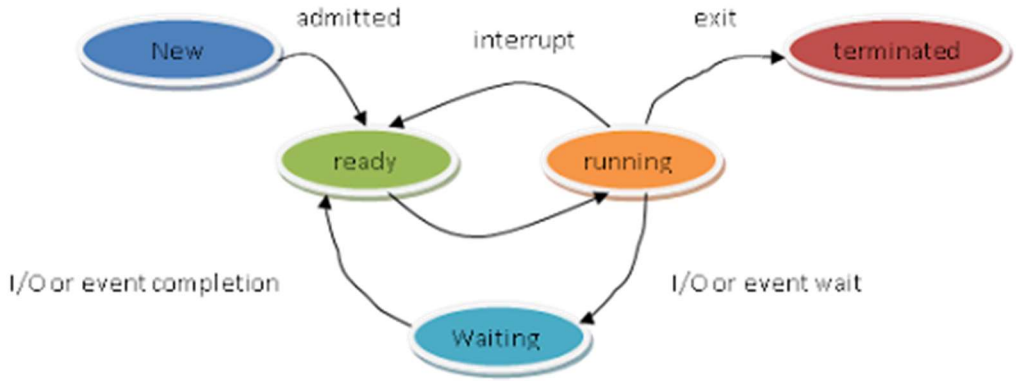
Q.1 (a) Explain in brief Program, Process and Thread.

Program:
In an operating system, a program is a set of instructions stored on secondary memory (hard disk, SSD, etc.). It is a passive entity and does not use CPU, memory, or other resources until it is executed. Example: a compiled C program or a Java application file.

Process:
A process is a program in execution. When the OS loads a program into main memory and starts its execution, it becomes a process. The operating system manages processes by allocating CPU time, memory, and other resources. Each process has its own address space, process ID (PID), and control information.

Thread:
A thread is the smallest unit of CPU execution in an operating system. It exists within a process and shares the process's memory, files, and resources. Threads are lightweight compared to processes and help achieve multitasking and parallelism within a single process, improving efficiency and system performance.

(b) What is Process State? Explain different states of a process with various queues generated at each stage.



Process State in Operating System

A **process state** represents the current condition or status of a process at a particular time during its life cycle. As a process executes, it changes its state according to the actions performed by the operating system, availability of CPU, and I/O operations. The OS maintains different **queues** to manage processes efficiently at each state.

Different States of a Process and Related Queues

1. **New State**
   This is the initial state of a process. When a process is created, it is in the *new* state.

The operating system is in the process of allocating resources and creating the process control block (PCB).
Queue: *New Queue* – holds newly created processes waiting to be admitted into the ready state.

2. **Ready State**
   In this state, the process is fully prepared to execute but is waiting for CPU allocation. The process has all required resources except the CPU.
   Queue: *Ready Queue* – contains all processes that are ready to execute and are waiting for CPU scheduling.

3. **Running State**
   When the CPU scheduler selects a process from the ready queue and assigns the CPU, the process enters the running state. In this state, the process instructions are being executed by the CPU.
   Queue: No queue (process is currently using the CPU).

4. **Waiting (Blocked) State**
   A process enters this state when it cannot continue execution until some event occurs, such as completion of an I/O operation or availability of a resource.
   Queue: *Waiting Queue / Device Queue* – holds processes waiting for I/O devices or specific events.

5. **Terminated (Exit) State**
   When a process finishes execution or is terminated by the OS or user, it enters the terminated state. All resources allocated to the process are released by the operating system.
   Queue: No queue (process is removed from the system).

(c) Give the functions of Operating System and Explain following OS in detail. 1. Time Sharing 2. Real Time 3. Batch Operating System

(c) Functions of Operating System and Types of Operating Systems

An **Operating System (OS)** is a collection of system programs that manages computer hardware and software resources and provides a convenient and efficient environment for execution of user programs. It acts as an interface between the user and the computer hardware.

Functions of Operating System

1. Process Management

Process management is one of the most important functions of an OS. A process is a program in execution. The operating system is responsible for creating and deleting processes, scheduling them for CPU execution, and controlling their synchronization and communication. It also handles issues such as deadlock detection and prevention to ensure smooth execution of multiple processes.

---

2. Memory Management

The operating system manages the main memory of the computer. It keeps track of which part of memory is used by which process and how much memory is free. The OS allocates memory to processes when required and deallocates it when the process finishes execution. Techniques such as paging and segmentation are used to improve memory utilization and system performance.

---

3. File Management

File management involves handling data stored on secondary storage devices. The OS provides mechanisms for creating, deleting, reading, writing, and organizing files and directories. It also manages file access permissions to protect data from unauthorized access.

---

4. Device Management

The OS manages all input and output devices connected to the computer system. It uses device drivers to communicate with hardware devices. The OS controls device allocation, scheduling of I/O requests, buffering, caching, and spooling to ensure efficient device utilization.

---

5. Secondary Storage Management

Secondary storage management includes managing disk space efficiently. The OS decides where data should be stored, keeps track of free disk space, and manages disk scheduling to improve performance.

---

6. Security and Protection

Security is a crucial function of the operating system. The OS protects system resources and user data from unauthorized access. It provides authentication mechanisms such as passwords and access control policies to ensure system safety.

---

7. User Interface

The OS provides a way for users to interact with the computer system. This can be in the form of a **Command Line Interface (CLI)** or a **Graphical User Interface (GUI)**. It simplifies communication between the user and the system.

Types of Operating Systems

---

1. Time Sharing Operating System

A **Time Sharing Operating System** allows multiple users to use the computer system simultaneously. The CPU time is divided into small units called **time slices** or **time quantum**. Each process is given a short amount of CPU time in a cyclic manner.

The OS switches rapidly between processes, giving the impression that all programs are running at the same time. Time sharing systems are designed to provide fast response time to users.

Characteristics

- Supports multi-user and multitasking environment
- Uses scheduling algorithms like Round Robin
- Provides interactive use of the system
- Short response time

Advantages

- Efficient utilization of CPU
- Immediate user response
- Suitable for interactive applications

Disadvantages

- Complex scheduling mechanism
- Performance may degrade if too many users are present

Examples

UNIX, Linux, Windows

2. Real Time Operating System (RTOS)

A **Real Time Operating System** is designed to respond to inputs and produce outputs within a specified time limit called a **deadline**. In RTOS, correctness depends on both the result and the time at which the result is produced.

RTOS is widely used in systems where timing is critical, such as medical equipment, aircraft control systems, and industrial automation.

Types of Real Time OS

- **Hard Real-Time OS:** Missing a deadline can cause system failure (e.g., missile guidance systems).
- **Soft Real-Time OS:** Missing deadlines reduces system performance but does not cause failure (e.g., multimedia systems).

Characteristics

- Predictable and deterministic behavior
- Very fast interrupt handling
- High reliability and stability

Advantages

- Guaranteed response time
- Suitable for time-critical applications

Disadvantages

- Expensive to develop and maintain
- Limited user interface and flexibility

Examples

VxWorks, QNX, RTLinux

---

3. Batch Operating System

A **Batch Operating System** is one of the earliest types of operating systems. In this system, jobs are collected into batches and processed one after another without user interaction.

Users submit their jobs to the computer operator, who groups similar jobs together. The OS executes them sequentially to improve system efficiency.

Characteristics

- No interaction between user and system during execution
- Jobs are executed in batches
- Uses job queues

Advantages

- High throughput
- Efficient for large volumes of similar jobs

Disadvantages

- Long turnaround time
- Difficult to debug errors
- No immediate feedback

Examples

IBM Mainframe Operating Systems

Q.2 (a) Define following with respect to CPU Scheduling Algorithms: 1. Turnaround Time 2.Waiting Time 3. Throughput

1. Turnaround Time
Turnaround time is the total time taken by a process from the moment it is submitted to the system until it completes its execution. It includes the time spent waiting in the ready queue, time spent in execution on the CPU, and time spent in I/O operations.
Turnaround Time = Completion Time − Arrival Time

---

2. Waiting Time
Waiting time is the total amount of time a process spends waiting in the ready queue for CPU allocation. It does not include the time during which the process is executing or performing I/O operations.
Waiting Time = Turnaround Time − Burst Time

---

3. Throughput
Throughput refers to the number of processes completed by the CPU in a given unit of time. It indicates how efficiently the CPU is being utilized.
Higher throughput means more processes are executed successfully in less time.

(b) Compare Round Robin, FCFS and Shortest Job First Algorithms for CPU Scheduling.

Basis of Comparison	FCFS (First Come First Serve)	SJF (Shortest Job First)	Round Robin (RR)
Basic Idea	Processes are executed in the order of their arrival in the ready queue.	Process with the shortest CPU burst time is executed first.	Each process is given CPU for a fixed time quantum in cyclic order.
Type of Algorithm	Non-preemptive	Can be preemptive or non-preemptive	Preemptive
CPU Allocation Method	Based on arrival time	Based on shortest burst time	Based on time slice (quantum)
Waiting Time	Generally high and unpredictable	Minimum average waiting time	Average waiting time is moderate
Turnaround Time	High	Minimum	Moderate
Response Time	Poor	Good	Very good
Starvation	No starvation	Possible starvation of long processes	No starvation
Fairness	Fair only in arrival order	Not fair to long jobs	Highly fair to all processes
Convoy Effect	Present	Not present	Not present
Complexity	Very simple to implement	Complex (requires burst time estimation)	Moderately complex
Context Switching	Very low	Low	High (frequent switching)
Suitable System	Batch operating systems	Batch systems where burst time is known	Time-sharing and multi-user systems
Main Advantage	Easy and simple scheduling	Optimal average waiting time	Good response time and fairness
Main Disadvantage	Poor performance and long wait	Difficult to predict burst time	Performance depends on time quantum

(c) What is Semaphore? Give the implementation of Readers-Writers Problem using Semaphore.

(c) Semaphore and Readers–Writers Problem using Semaphore

What is a Semaphore?

A **Semaphore** is a synchronization tool used in an operating system to control access to shared resources by multiple processes or threads. It helps prevent **race conditions** and ensures **mutual exclusion** when processes execute concurrently.

A semaphore is basically an **integer variable** that can be accessed only through two atomic operations:

- **wait() / P()** → Decreases the semaphore value
- **signal() / V()** → Increases the semaphore value

Types of Semaphore

1. **Binary Semaphore**
   - Value is either 0 or 1
   - Used for mutual exclusion (similar to a mutex)
2. **Counting Semaphore**
   - Value can be any non-negative integer
   - Used when multiple instances of a resource are available

---

Semaphore Operations (Conceptual)

- **wait(S):**
  - If `S > 0`, decrement `S` and continue
  - If `S = 0`, process is blocked
- **signal(S):**
  - Increment `S`
  - Wake up one blocked process (if any)

Readers–Writers Problem

The **Readers–Writers Problem** is a classic synchronization problem in operating systems. It deals with processes that want to **read** or **write** shared data.

Rules

- Multiple readers can read the data **simultaneously**
- Only one writer can write at a time
- While a writer is writing, **no reader** should read

Goal

To allow:

- Concurrent reading
- Exclusive writing
- Proper synchronization using semaphores

Semaphore-Based Solution (Readers Priority)

Shared Variables

```
int readCount = 0;
semaphore mutex = 1;       // Protects readCount
semaphore wrt = 1;         // Controls access to shared resource
```

Reader Process Implementation

```
Reader()
{
    wait(mutex);           // Lock readCount
    readCount++;
    if (readCount == 1)
        wait(wrt);         // First reader blocks writers
    signal(mutex);         // Unlock readCount

    // --- Reading Section ---
    read data;

    wait(mutex);
    readCount--;
    if (readCount == 0)
        signal(wrt);       // Last reader allows writers
    signal(mutex);
}
```

Explanation (Reader)

- First reader locks the writer semaphore
- Multiple readers can read simultaneously
- Last reader releases the writer semaphore

Writer Process Implementation

```
Writer()
{
    wait(wrt);             // Lock resource for writing

    // --- Writing Section ---
    write data;

    signal(wrt);           // Release resource
}
```

Explanation (Writer)

- Writer gets exclusive access
- No reader or writer can access data while writing

Advantages of Semaphore Solution

- Prevents race conditions
- Allows maximum parallelism for readers
- Ensures data consistency

Disadvantages

- Writers may suffer starvation if readers keep coming
- More complex than simple mutual exclusion

OR

(c) What is Monitor? Give the implementation of Bounded Buffer Producer Consumer Problem using Monitor.

(c) Monitor and Bounded Buffer Producer–Consumer Problem using Monitor

---

What is a Monitor?

A **Monitor** is a high-level synchronization construct used in operating systems to control access to shared resources. It ensures **mutual exclusion automatically**, meaning that **only one process or thread can be active inside a monitor at a time**.

Unlike semaphores, monitors combine:

- **Shared data**
- **Operations (procedures)**
- **Synchronization**

into a single logical unit, making synchronization **simpler and safer**.

Key Features of Monitor

- Mutual exclusion is automatic
- Uses **condition variables** for synchronization
- Avoids busy waiting
- Easier to use than semaphores

Condition Variables

Condition variables are used for process synchronization inside a monitor:

- `wait()` → Process waits until condition becomes true
- `signal()` → Wakes up a waiting process

Bounded Buffer Producer–Consumer Problem

The **Producer–Consumer Problem** involves two types of processes:

- **Producer**: Produces items and puts them into a buffer
- **Consumer**: Consumes items from the buffer

The buffer has a **fixed size**, so:

- Producer must wait if the buffer is full
- Consumer must wait if the buffer is empty

Monitor-Based Solution

Monitor Structure

```
monitor BoundedBuffer
{
    int buffer[N];
    int count = 0, in = 0, out = 0;
    condition notFull, notEmpty;
```

Producer Implementation

```
    procedure produce(item x)
    {
        if (count == N)
            notFull.wait();     // Wait if buffer is full

        buffer[in] = x;
        in = (in + 1) % N;
        count++;

        notEmpty.signal();      // Signal consumer
    }
```

Consumer Implementation

```
    procedure consume()
    {
        if (count == 0)
            notEmpty.wait();    // Wait if buffer is empty

        item x = buffer[out];
        out = (out + 1) % N;
        count--;

        notFull.signal();       // Signal producer
        return x;
```

```
    }
}
```

Explanation

- The **monitor ensures mutual exclusion** automatically
- Producer waits when buffer is full using `notFull.wait()`
- Consumer waits when buffer is empty using `notEmpty.wait()`
- `signal()` wakes up waiting producer or consumer

Advantages of Monitor Solution

- Simple and clean synchronization
- No explicit lock management
- Reduces programming errors
- Avoids race conditions

Disadvantages

- Language support required (Java, C#, etc.)
- Less flexible than semaphores in low-level systems

A **Monitor** is an advanced synchronization mechanism that simplifies concurrent programming. The **Bounded Buffer Producer–Consumer Problem** can be efficiently solved using monitors by combining shared data, synchronization, and mutual exclusion in a single construct, making the solution safe and easy to understand.

Q.3 (a) What is Shell and Kernel in Unix?

In the **UNIX operating system**, the **Shell** and **Kernel** are two major components that work together to provide interaction between the user and the system hardware.

Kernel

The **Kernel** is the **core** of the UNIX operating system. It is the first program loaded when the system starts and remains in memory throughout the system's operation. The kernel directly interacts with the hardware and manages all system resources.

Functions of Kernel

- Manages **CPU scheduling** and process execution
- Handles **memory management** and allocation
- Controls **file system** operations
- Manages **device drivers** and I/O operations
- Provides **security and protection**
- Handles system calls from user programs

The kernel operates in **privileged mode**, which ensures that critical system resources are protected from unauthorized access.

Shell

The **Shell** is a **command interpreter** that acts as an interface between the user and the kernel. It accepts commands entered by the user, interprets them, and requests the kernel to execute them.

Functions of Shell

- Reads and interprets user commands
- Executes system programs and utilities
- Provides scripting capability
- Handles input/output redirection and piping
- Offers environment customization

The shell runs in **user mode** and communicates with the kernel using system calls.

Types of UNIX Shells

- **Bourne Shell (sh)**
- **C Shell (csh)**
- **Korn Shell (ksh)**
- **Bash (Bourne Again Shell)**

(b) List the necessary conditions that lead to deadlock.

A **deadlock** is a critical situation in an operating system where a group of processes become permanently blocked because each process is waiting for a resource that is held by another process in the same group. As a result, none of the processes can proceed. For a deadlock to occur, **four necessary conditions must exist simultaneously**. These conditions are known as **Coffman's conditions**.

1. Mutual Exclusion

The **mutual exclusion** condition states that at least one resource must be **non-shareable**, meaning only one process can use that resource at a time. If another process requests the same resource, it must wait until the resource is released.

Explanation:
Some resources, such as printers, scanners, and tape drives, cannot be shared by прием multiple processes simultaneously. When a process holds such a resource, other processes requesting it are blocked.

Example:
If Process P1 is using a printer, Process P2 cannot use the printer until P1 releases it.

---

2. Hold and Wait

According to the **hold and wait** condition, a process must be **holding at least one resource** while **waiting for additional resources** that are currently allocated to other processes.

Explanation:
This condition allows a process to lock some resources and then wait indefinitely for other resources, creating a situation where resource allocation becomes interdependent.

Example:
Process P1 holds Resource R1 and waits for Resource R2, while Process P2 holds Resource R2 and waits for Resource R1.

---

3. No Preemption

The **no preemption** condition means that resources cannot be forcibly taken away from a process. A resource can only be released voluntarily by the process after it has completed its task.

Explanation:
The operating system does not have the authority to interrupt a process and take back its allocated resources, even if other processes are waiting.

Example:
If a process is using a printer, the OS cannot forcibly remove the printer from the process until printing is complete.

---

4. Circular Wait

The **circular wait** condition occurs when a set of processes forms a circular chain, where each process is waiting for a resource held by the next process in the chain.

Explanation:
This creates a closed loop of dependency among processes, making it impossible for any process to proceed.

Example:

- P1 waits for a resource held by P2
- P2 waits for a resource held by P3
- P3 waits for a resource held by P1

This circular dependency results in deadlock.

(c) Explain Banker's Algorithm for Deadlock Avoidance with illustration

YOUTUBE LINK: https://youtu.be/7gMLNiEz3nw?si=Pydm1zcYMrvmaAYW

What is Banker's Algorithm?

Banker's Algorithm is a **deadlock avoidance algorithm** used in operating systems. It is mainly applied in systems where the **maximum resource requirement of each process is known in advance**.
The algorithm ensures that the system **always remains in a safe state** before allocating resources.

It is called *Banker's Algorithm* because it works like a banker who gives loans only if he is sure that all customers can get their maximum loan and still the bank will not go bankrupt.

Objectives of Banker's Algorithm

- Avoid deadlock
- Allocate resources safely
- Check whether a system is in a **safe state** or **unsafe state**

Data Structures Used

1. **Allocation Matrix**
   Shows the number of resources currently allocated to each process.
2. **Max Matrix**
   Shows the maximum number of resources each process may need.
3. **Available Vector**
   Shows the number of available instances of each resource.
4. **Need Matrix**
   Shows the remaining resource requirement of each process.

Need = Max − Allocation

Illustration (Based on Given Data)

Given Resource Types: A, B, C

Process Information

Process	Allocation (A B C)	Max (A B C)
P0	2 1 0	8 6 3
P1	1 2 2	9 4 3
P2	0 2 0	5 3 3
P3	3 0 1	4 2 3

Available Resources

```
Available = (4, 3, 2)
```

Step 1: Calculate Need Matrix

Process	Need (A B C)
P0	6 5 3
P1	8 2 1
P2	5 1 3
P3	1 2 2

(Need = Max − Allocation)

Step 2: Safety Algorithm Execution

Initial:

```
Work = (4, 3, 2)
Finish = false for all processes
```

Check P3

Need (1,2,2) ≤ Work (4,3,2)
Execute P3, release its allocation:

```
Work = (4,3,2) + (3,0,1) = (7,3,3)
Finish[P3] = true
```

Check P2

Need (5,1,3) ≤ Work (7,3,3)
Execute P2:

```
Work = (7,3,3) + (0,2,0) = (7,5,3)
Finish[P2] = true
```

Check P0

Need (6,5,3) ≤ Work (7,5,3)
Execute P0:

```
Work = (7,5,3) + (2,1,0) = (9,6,3)

Finish[P0] = true
```

Check P1

Need (8,2,1) ≤ Work (9,6,3)

Execute P1:

Work = (9,6,3) + (1,2,2) = (10,8,5)

Finish[P1] = true

Safe Sequence

$$\boxed{P3 \rightarrow P2 \rightarrow P0 \rightarrow P1}$$

Since **all processes can finish**, the system is in a **SAFE STATE**.

Key Points of Banker's Algorithm

- Ensures deadlock avoidance
- Requires advance knowledge of maximum resource needs
- Works only if system is in a safe state
- Uses safety and resource request algorithms

Advantages

- Prevents deadlock completely
- Guarantees system safety
- Efficient for controlled environments

Disadvantages

- Requires prior knowledge of resource needs
- Not suitable for real-time systems
- High overhead for large systems

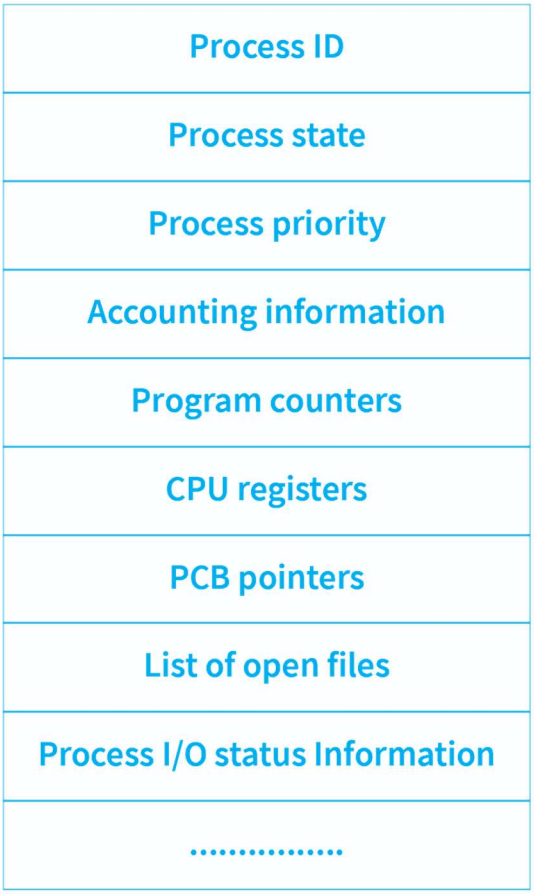
Banker's Algorithm is a powerful deadlock avoidance technique that checks system safety before granting resource requests. By ensuring a **safe sequence**, it prevents deadlock and guarantees smooth execution of all processes.

OR

Q.3 (a) Give the difference between Deadlock and Starvation.

Basis	Deadlock	Starvation
Meaning	Processes are permanently blocked due to circular wait	Process waits indefinitely due to lack of resources
Cause	Circular dependency among processes	Continuous allocation to higher priority processes
Number of Processes	Involves two or more processes	May involve a single process
Resource Dependency	Mutual dependency among processes	No circular dependency
Resolution	Requires deadlock handling techniques	Can be avoided using aging
System State	System comes to a halt for those processes	System continues to run
Prevention	Avoid by breaking deadlock conditions	Avoid by fair scheduling

(b) What is Process Control Block (PCB)?Explain various entries in PCB.



What is Process Control Block (PCB)?

A **Process Control Block (PCB)** is a data structure maintained by the **Operating System** for each process. It stores **all the information required to manage and control a process**. Whenever a process is created, the OS creates a PCB for it, and when the process terminates, its PCB is deleted.

The PCB plays a vital role during **process scheduling, context switching, and resource management**. When the CPU switches from one process to another, the OS saves the current process's state in its PCB and loads the next process's state from its PCB.

Importance of PCB

- Helps OS keep track of all active processes
- Stores execution state during context switching
- Maintains process-related information efficiently
- Essential for multitasking and multiprogramming

Various Entries (Fields) in Process Control Block

The PCB contains several important entries, each serving a specific purpose:

---

1. Process ID (PID)

- A **unique identification number** assigned to each process.
- Helps the operating system distinguish between different processes.
- Used for process management and scheduling.

---

2. Process State

- Indicates the current state of the process.
- Possible states include: **New, Ready, Running, Waiting (Blocked), and Terminated**.
- Helps the OS decide what action to take for the process.

---

3. Process Priority

- Stores the priority level of the process.
- Used by the CPU scheduler to decide which process should be executed next.
- Higher priority processes are generally executed before lower priority ones.

---

4. Program Counter

- Contains the **address of the next instruction** to be executed by the process.
- During context switching, the program counter value is saved and restored so that the process can resume execution correctly.

---

5. CPU Registers

- Stores the contents of all CPU registers (such as accumulator, general-purpose registers, stack pointer).
- Necessary to restore the process's execution state after interruption or context switching.

---

6. Memory Management Information

- Contains information about memory allocation to the process.
- May include base and limit registers, page tables, or segment tables.
- Helps the OS manage and protect process memory space.

---

7. Accounting Information

- Stores data such as:
  - CPU time used
  - Time limits
  - Process start time
- Useful for system performance analysis, billing, and auditing.

---

8. I/O Status Information

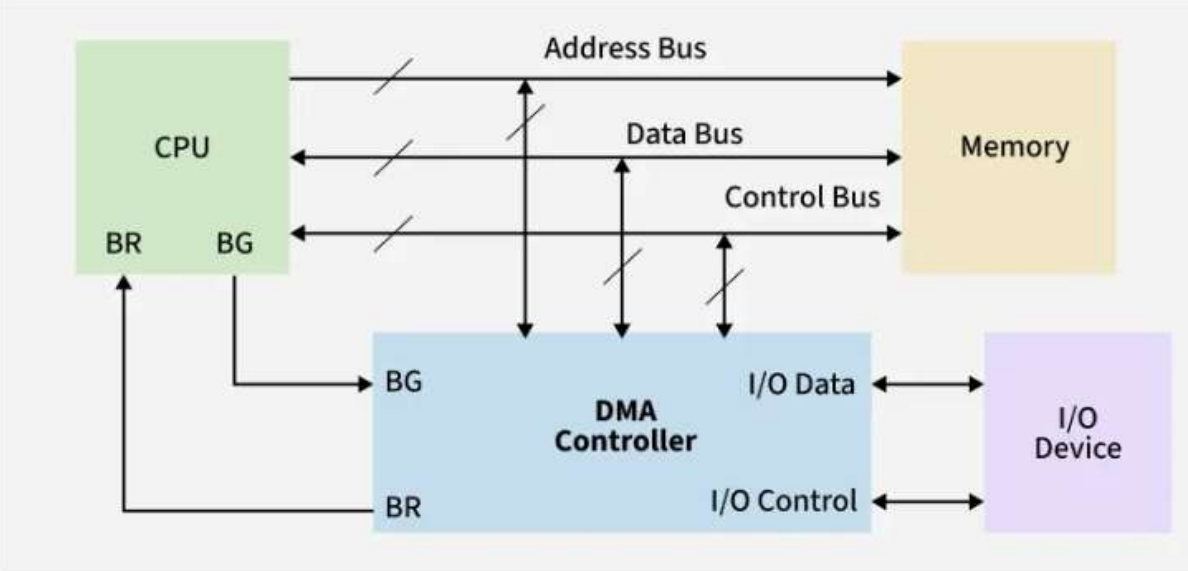
- Keeps track of I/O devices allocated to the process.
- Contains a **list of open files**, pending I/O requests, and device status.
- Helps the OS manage input/output operations efficiently.

---

9. PCB Pointers

- Contains pointers to other PCBs.
- Used to maintain **process queues** such as ready queue, waiting queue, etc.
- Helps the OS in scheduling and queue management.

(c) Draw the block diagram for DMA. Write steps for DMA data transfer.



DMA stands for **Direct Memory Access**. It is a technique that allows **I/O devices (like keyboards, disk drives, or printers)** to send or receive data **directly to or from the memory**, without involving the CPU for each byte of data.

- Normally, the CPU handles all data transfers between memory and I/O devices.
- With DMA, the **DMA controller** does the data transfer while the CPU can do other tasks.
- This makes data transfer **faster** and **more efficient**, especially for large blocks of data.

Step 1: CPU Initiates the Process

The process of DMA begins when an I/O device (like a hard disk, keyboard, or network card) needs to transfer data to memory or read data from memory. Unlike the traditional method where the CPU manages every byte of data transfer, DMA allows the device to directly communicate with memory to improve speed.

Here's what happens:

1. **CPU Sets Up DMA Controller:**
   The CPU is responsible for initializing the DMA controller before any transfer happens. It does not transfer the data itself but prepares the DMA controller with all the required details.
2. **Information Provided to DMA Controller:**
   - **Memory Address:** The exact location in memory where the data should be stored (for device → memory transfer) or from where the data should be read (for memory → device transfer).
   - **Data Size:** The number of bytes that need to be transferred. This allows DMA to know when the transfer is complete.
   - **Transfer Direction:** Whether the operation is a read from the device to memory or a write from memory to the device.

At this stage, the CPU has done its part by configuring the DMA controller and can step aside from the actual data transfer.

Step 2: DMA Requests Control of the Bus

Once the DMA controller is ready to start transferring data, it must gain access to the system bus. The system bus is a shared communication pathway used by the CPU, memory, and I/O devices, so only one can use it at a time.

1. **Bus Request (BR):**
   The DMA controller sends a **Bus Request (BR)** signal to the CPU. This signal essentially says, "CPU, I need control of the address and data buses to transfer data directly."

2. **Waiting for CPU Response:**
   The DMA controller cannot start the transfer immediately. It waits until the CPU finishes its current operation on the bus so that bus conflicts do not occur.

This step ensures that the DMA controller gets exclusive access to the bus without interfering with the CPU's ongoing operations.

Step 3: CPU Grants Bus Control

After receiving the bus request from the DMA controller, the CPU will eventually respond:

1. **Bus Grant (BG):**
   The CPU sends a **Bus Grant (BG)** signal to the DMA controller. This means: "Okay, DMA, you can use the buses now."
2. **CPU Status During DMA Transfer:**
   While the DMA controller has control of the bus, the CPU either remains idle for bus-related operations or continues executing instructions that **do not require the bus**. This allows the CPU to multitask while the DMA handles the data transfer in the background.

This step is crucial because it prevents the CPU and DMA from trying to use the bus at the same time, which could lead to data corruption.

Step 4: DMA Performs Data Transfer

Now comes the core of the DMA operation—the actual data transfer. The DMA controller directly communicates with memory and the I/O device without involving the CPU. This is what makes DMA fast and efficient.

1. **Memory Address Placement:**
   The DMA controller places the memory address on the address bus. This tells the memory where to read from or write to.
2. **Data Transfer:**
   - **Device → Memory (Read):** DMA reads data from the I/O device and writes it directly into memory.
   - **Memory → Device (Write):** DMA reads data from memory and writes it directly to the I/O device.
3. **Control Signals:**
   DMA sends necessary signals through the control bus to coordinate the operation, such as **read**, **write**, or **acknowledge signals**.
4. **CPU Bypassed:**
   The CPU is completely bypassed during this transfer. This reduces CPU overhead significantly, allowing it to perform other tasks or computations.

This step is repeated automatically for each byte (or block) of data until the entire requested data transfer is complete.

Step 5: DMA Updates and Completes Transfer

As the DMA controller works, it keeps track of the transfer:

1. **Byte Count Tracking:**
   The DMA controller keeps a counter to track how many bytes have already been transferred.
2. **Memory Address Update:**
   After transferring each byte (or word), the DMA increments the memory address to point to the next location for the following transfer.
3. **Completion:**
   When all the bytes are transferred successfully, the DMA controller releases the bus.

At this point, the actual transfer is complete, and the system is ready for the CPU to resume control fully.

Step 6: DMA Notifies CPU

After the data transfer is complete, the DMA controller sends an **interrupt signal** to the CPU. This serves as a notification that:

1. The DMA operation is finished.
2. The data in memory or at the I/O device is now ready for processing.

The CPU responds to this interrupt and resumes any processing that depends on the newly transferred data.

Q.4 (a) Explain Internal and External Fragmentation in Memory Management.

Memory Management and Fragmentation

In operating systems, **memory management** is responsible for efficiently allocating and deallocating memory to processes. Fragmentation occurs when memory is wasted, even though there may be enough total memory available, it is not usable effectively due to the way memory is allocated.

There are **two main types of fragmentation**: **Internal** and **External**.

---

1. Internal Fragmentation

Definition:
Internal fragmentation occurs when **fixed-sized memory blocks** are allocated to processes,

and the process does not use the entire block. The unused memory within the allocated block is wasted.

Explanation:

- Memory is divided into fixed-sized blocks (like pages or partitions).
- A process is allocated a block that is **equal to or slightly larger** than its required memory.
- If the process does not need the full block, the remaining memory inside the block **remains unused**.
- This wasted memory inside the allocated block is called **internal fragmentation**.

Example:

- Suppose memory is divided into blocks of **4 KB** each.
- A process requires **3 KB** of memory.
- It is allocated a 4 KB block, so **1 KB is wasted inside the block**.
- This 1 KB is **internal fragmentation**.

Key Points:

- Occurs due to **fixed-sized allocation**.
- Wasted memory is **inside the allocated block**.
- Common in **paging** or **fixed partition allocation** schemes.

---

2. External Fragmentation

Definition:
External fragmentation occurs when **free memory is scattered in small pieces** between allocated blocks. Even though the total free memory is enough to satisfy a request, it cannot be allocated because it is **not contiguous**.

Explanation:

- In **dynamic memory allocation**, processes of different sizes are loaded and removed from memory.
- Over time, memory gets broken into small **non-contiguous free blocks**.
- When a new process arrives, it may require a large contiguous block, but no such block exists, even though the total free memory is sufficient.
- This wasted, unusable memory outside allocated blocks is called **external fragmentation**.

Example:

- Total memory: 16 KB
- Free memory after some allocations: 2 KB, 3 KB, 4 KB, 2 KB (scattered)
- A process requires 5 KB.
- Total free memory = 11 KB > 5 KB, but no single block ≥ 5 KB is available.

- This is **external fragmentation**.

Key Points:

- Occurs due to **variable-sized allocation**.
- Wasted memory is **outside allocated blocks**, scattered across memory.
- Can be reduced using **compaction**, which shifts allocated memory to combine free blocks.

(b) Explain the Linux/Unix Commands: cat, rmdir, sort, chmod.

---

1. `cat` Command

The `cat` command in Linux/Unix is used to **display the contents of a file, create a new file, or combine multiple files**. It is a simple and quick way to view or merge text files.

Examples:

1. View a file:

```
cat file1.txt
```

2. Combine files into a new file:

```
cat file1.txt file2.txt > file3.txt
```

---

2. `rmdir` Command

The `rmdir` command is used to **delete empty directories**. It cannot remove directories that contain files or subdirectories.

Examples:

1. Remove an empty directory:

```
rmdir myfolder
```

2. Remove multiple empty directories:

```
rmdir folder1 folder2
```

---

3. `sort` Command:

The `sort` command is used to arrange the lines of a text file or input in order**, either alphabetically or numerically. It helps organize data for easy reading.**

Examples:

1. Sort alphabetically:

```
sort names.txt
```

2. Sort numerically:

```
sort -n numbers.txt
```

---

4. `chmod` Command

The `chmod` command is used to **change the permissions of files or directories**, determining who can read, write, or execute them.

Examples:

1. Give full permissions to owner, read and execute to group and others:

```
chmod 755 file.txt
```

2. Remove write permission for others:

```
chmod o-w file.txt
```

(c) Explain FIFO, LRU and Optimal Page Replacement Algorithms with example.

For better explanation and example:

https://youtu.be/8rcUs5RutX0?si=BQQVDi9-UnGUXW3g

https://youtu.be/dYIoWkCvd6A?si=Y3J47ZYlQYOx1k3V

https://youtu.be/q2BpMvPhhrY?si=XVQp6x_uVh1bTjVZ

1. FIFO (First-In-First-Out) Page Replacement

FIFO is the simplest page replacement algorithm. It works like a **queue**, where the **oldest page in memory is replaced first** when a new page needs to be loaded. The CPU maintains the order in which pages were brought into memory. When a page fault occurs, the page that came first is removed, regardless of how frequently or recently it was used. It is easy to implement but may replace pages that are still in use, causing more page faults in some cases.

- Works like a **queue (first come, first served)**.

- The CPU keeps track of the order in which pages are loaded into memory.

- When a new page needs memory and all frames are full, **the oldest page (the one that entered first) is replaced**.

- It does not consider how often or recently a page is used.

---

2. LRU (Least Recently Used) Page Replacement

LRU replaces the page that has **not been used for the longest period of time**. It keeps track of page usage over time, either through counters or stacks. When a page fault occurs, the **least recently accessed page** is removed from memory. This method assumes that pages used recently are likely to be used again soon, making it more efficient than FIFO. It requires additional hardware or software mechanisms to track usage.

- Works based on **recent usage of pages**.

- The CPU keeps track of **when each page was last used**.

- When a page fault occurs and memory is full, **the page that has not been used for the longest time is replaced**.

- Assumes that recently used pages are likely to be used again soon.

---

3. Optimal Page Replacement

The Optimal algorithm replaces the page that **will not be used for the longest period in the future**. It minimizes page faults because it always chooses the page whose removal will affect the system least. In practice, it is impossible to implement precisely because it requires knowledge of future memory references. However, it is used as a **benchmark** to compare the performance of other algorithms.

- Works based on **future knowledge of page references**.

- When a page fault occurs and memory is full, **the page that will not be used for the longest time in the future is replaced**.

- Minimizes the number of page faults.

- Cannot be implemented in real systems because future references are unknown; used mainly for comparison.

OR

Q.4 (a) Write a shell script to find a factorial of given number n.

```
# Read a number from the user
echo "Enter a number:"
read n

# Initialize factorial to 1
factorial=1

# Check if the number is negative
if [ $n -lt 0 ]; then
    echo "Factorial is not defined for negative numbers."
else
    # Loop from 1 to n and multiply
    for (( i=1; i<=n; i++ ))
    do
        factorial=$((factorial * i))
    done
    echo "Factorial of $n is: $factorial"
fi
```

(b) Compare Paging and Segmentation with reference to Memory Managament.

Feature	Paging	Segmentation
Definition	Paging divides the logical memory of a process into **fixed-size pages** and physical memory into **equal-size frames**. Pages are mapped to frames allowing **non-contiguous allocation**.	Segmentation divides the logical memory of a process into **variable-sized segments** according to program structure, such as **code, data, stack, etc.** Each segment is treated as a logical unit.
Memory Division	Fixed-size blocks called pages.	Variable-size blocks called segments.
Fragmentation	May cause **internal fragmentation** because the last page may not be fully used.	May cause **external fragmentation** as segments are of different sizes.
Addressing	Logical addresses are split into **page number** and **page offset**.	Logical addresses are split into **segment number** and **segment offset**.
Memory Allocation	Non-contiguous memory allocation is possible, making it easy to utilize memory efficiently.	Memory allocation is contiguous per segment, which may lead to unused holes.
Ease of Management	Simple to manage due to fixed page size; mapping can be done using a **page table**.	More complex to manage because segments are of variable size; requires a **segment table**.
Logical View	Provides a **linear view** of memory; processes see memory as a sequence of pages.	Provides a **logical view** matching program structure; each segment represents a meaningful unit.
Use Case	Used in modern OS for **virtual memory management**; avoids external fragmentation.	Useful when program structure is important, such as **separate code, data, stack segments**.

(c) Explain various Disk Arm Scheduling Algorithms for Disk Space Management.

For examples: https://youtu.be/9uoa_p8q47Y?si=-wCX9xSH9IOPfZpz

1. FCFS (First-Come, First-Served)

How it works:

- Requests are processed in the **order they arrive**.
- The disk arm services the first request in the queue, then moves to the next, and so on.

Advantages:

- Simple to implement.
- Fair, as all requests are treated equally.

Disadvantages:

- Can result in **long average seek time** if requests are scattered.

---

2. SSTF (Shortest Seek Time First)

How it works:

- The disk arm selects the request **closest to its current position**.
- Reduces the total distance the arm moves compared to FCFS.

Advantages:

- Reduces **average seek time**.

Disadvantages:

- Can cause **starvation** for requests far from the current arm position.

---

3. SCAN (Elevator Algorithm)

How it works:

- Disk arm moves in one direction servicing requests **until it reaches the end**, then reverses direction.
- Works like an elevator moving up and down.

Advantages:

- More **uniform wait time** than SSTF.
- Avoids long delays for requests at the extremes.

Disadvantages:

- Requests may wait longer if the arm is moving in the opposite direction.

---

4. C-SCAN (Circular SCAN)

How it works:

- Disk arm moves in **one direction only**.
- After reaching the end, it **jumps to the beginning** without servicing requests on the return.

- Provides a **more uniform response time** than SCAN.

Advantages:

- Fairer than SCAN; all requests wait roughly the same time.

Disadvantages:

- Slightly longer total seek distance due to jump back to start.

---

5. LOOK and C-LOOK

How it works:

- **LOOK:** Disk arm only goes as far as the **last request in each direction** instead of going to the end of the disk.
- **C-LOOK:** Circular version of LOOK; after reaching the last request, arm jumps to the first request without going to the beginning of the disk.

Advantages:

- Reduces unnecessary arm movement compared to SCAN and C-SCAN.
- Improves efficiency.

Disadvantages:

- Slightly more complex to implement than SCAN and FCFS.

Q.5 (a) Explain difference between Security and Protection?

Feature	Security	Protection
Definition	Security refers to the mechanisms and policies used to **prevent unauthorized access, misuse, or damage** to the computer system and its resources.	Protection refers to the mechanisms and techniques used to **control access to resources by legitimate users or processes** within the system.
Purpose	Ensures **overall system safety** from external and internal threats.	Ensures **correct usage of resources** by authorized processes and users.
Scope	Broader in scope; includes threats from outside (hackers, malware) and inside (malicious users, software bugs).	Narrower in scope; focuses on **regulating access rights and permissions**.

Feature	Security	Protection
Focus	Focuses on **preventing attacks and breaches** that compromise system integrity, confidentiality, and availability.	Focuses on **enforcing access rules** and maintaining controlled access to system resources.
Mechanism	Implemented using **firewalls, encryption, antivirus, authentication, intrusion detection** systems.	Implemented using **access control lists (ACLs), user IDs, passwords, file permissions, capability lists**.
Concerned With	Protecting the system from **unauthorized users** or external threats.	Protecting resources from **misuse by legitimate users** or processes.
Examples	Encryption to protect data, antivirus to prevent malware, authentication mechanisms.	File permissions (read, write, execute), process access rights, memory protection.
Goal	Maintain **confidentiality, integrity, and availability** of system and data.	Maintain **correctness and controlled access** of system resources.
Implementation Level	Often implemented at **system-wide or network level**.	Often implemented at **operating system or resource level**.
Dependency	Depends on protection mechanisms but also includes additional safeguards against threats.	Provides the basis for security by **defining what each user/process is allowed to do**.
Example Scenario	Preventing a hacker from stealing sensitive files over the network.	Ensuring that only the owner of a file can modify it, while others can only read it.

(b) What are the Allocation Methods of a Disk Space?

Disk Space Allocation Methods

Disk space allocation refers to how **files are stored on a disk**. There are **three main methods**:

---

1. Contiguous Allocation

Definition:

- The file is stored in **consecutive blocks** on the disk.
- Each file occupies a set of continuous disk blocks.

Characteristics:

- Simple to implement.
- **Fast access** because the blocks are sequential.
- Difficult to **grow files dynamically**; can cause **external fragmentation**.

Example:

- A file needing 5 blocks might occupy blocks 10–14 on the disk.

---

2. Linked Allocation

Definition:

- Each file is stored as a **linked list of disk blocks**.
- Each block contains a pointer to the **next block** of the file.

Characteristics:

- No external fragmentation.
- Files can **grow dynamically**.
- Sequential access is fast, but **random access is slow**.

Example:

- File blocks: 5 → 8 → 12 → 20, each pointing to the next block.

---

3. Indexed Allocation

Definition:

- Each file has an **index block** that contains pointers to all the blocks of the file.
- All file blocks can be anywhere on the disk.

Characteristics:

- Supports **direct access** efficiently.
- No external fragmentation.
- Slight overhead for storing index blocks.

Example:

- Index block contains: 3, 7, 9, 15 → pointing to actual file blocks.

(c) Explain in Detail File and Directory Management of Unix/Linux Operating System in detail.

File and Directory Management in Unix/Linux Operating System

Unix/Linux operating systems provide a **well-structured file and directory management system** that allows users and processes to store, retrieve, and manipulate data efficiently. Unix treats **everything as a file**, including devices, pipes, and directories. This design simplifies management and provides a uniform interface for accessing system resources.

---

1. File Management in Unix/Linux

Definition:
A file is a **logical storage unit** that contains data, information, or instructions. Files in Unix/Linux can be **regular files, directories, special files, or pipes**. Every file has associated metadata and attributes stored in an **inode (index node)**, which contains details about the file but not its name.

Types of Files in Unix/Linux

1. **Regular files:** Contain user or application data such as text files, program files, or multimedia files.
2. **Directory files:** Special files that store information about other files, including names and pointers to inodes.
3. **Special files (device files):** Represent hardware devices, divided into:
    - **Character devices** (e.g., keyboard, mouse)
    - **Block devices** (e.g., hard drives, USB storage)
4. **Pipe files and sockets:** Used for **inter-process communication** (IPC) to allow data transfer between processes.

File Attributes (Stored in Inode)

- File type (regular, directory, special)
- Permissions (read, write, execute) for owner, group, and others
- Owner user ID (UID) and group ID (GID)
- File size in bytes
- Creation, modification, and last access timestamps
- Number of links (hard links pointing to the file)
- Pointers to the actual data blocks on disk

File Operations

Unix/Linux provides several **basic file operations** that allow users and programs to manipulate files:

- **Creation:** `touch filename`, `echo > filename`

- **Deletion:** `rm filename`
- **Reading:** `cat`, `less`, `more`, `head`, `tail`
- **Writing and Editing:** `echo "text" > filename`, `nano filename`, `vim filename`
- **Renaming and Moving:** `mv oldname newname`
- **Copying:** `cp source dest`
- **Changing permissions and ownership:** `chmod`, `chown`, `chgrp`

Unix/Linux **supports multiple users** and enforces access control using file permissions to ensure security and protection of data.

---

2. Directory Management in Unix/Linux

Definition:
A directory is a **special type of file** that stores information about other files and directories. It provides a **hierarchical structure** to organize files, enabling efficient file management, search, and access.

Directory Structure

- **Hierarchical (tree-like) structure:** Modern Unix/Linux systems use a tree hierarchy, starting from the **root directory** `/`.
- Directories can contain files and subdirectories, allowing a nested organizational structure.

Types of Directory Structures

1. **Single-level directory:** All files are stored in a single directory. Simple but impractical for multiple users.
2. **Two-level directory:** Each user has a separate directory to avoid file name conflicts.
3. **Tree-structured directory:** A hierarchical arrangement where directories can contain subdirectories, supporting multiple users efficiently.
4. **Acyclic graph / general graph directories:** Allows links (hard links and soft links) between files and directories to enable shared access.

Directory Operations

- **Creating a directory:** `mkdir dirname`
- **Removing a directory:** `rmdir dirname` (only if empty)
- **Listing contents:** `ls -l`, `ls -a`
- **Changing directory:** `cd dirname`
- **Absolute vs Relative paths:**
  - **Absolute path:** Begins from the root directory `/`, e.g., `/home/user/docs`
  - **Relative path:** Begins from the current working directory, e.g., `docs/file.txt`

Special Directory Entries

- `.` → Current directory
- `..` → Parent directory
- `/` → Root directory

---

3. File and Directory Permissions

Unix/Linux uses **permissions and ownership** to control access to files and directories. Every file/directory has permissions divided among **owner, group, and others**:

- **Read (r):** Allows viewing file contents or listing directory contents
- **Write (w):** Allows modifying file contents or creating/deleting files in a directory
- **Execute (x):** Allows executing a file as a program or accessing a directory

Permissions can be modified using commands: `chmod` (change permissions), `chown` (change owner), `chgrp` (change group).

Example:

- `chmod 755 file.txt` → Owner can read, write, execute; group and others can read and execute.

---

4. Hierarchical File System in Unix/Linux

The **Unix/Linux file system** is hierarchical, starting from the **root directory** `/`. Important directories include:

- `/home` → User home directories
- `/bin` → Essential binaries and commands
- `/etc` → Configuration files
- `/tmp` → Temporary files
- `/dev` → Device files
- `/usr` → User applications and data

The hierarchical structure allows **efficient storage, navigation, and management** of files for both users and processes.

---

5. Advantages of Unix/Linux File and Directory Management

1. **Uniformity:** Treats everything as a file, simplifying access to resources.
2. **Security:** Permissions and ownership control access to files and directories.
3. **Flexibility:** Supports hierarchical structures for better organization.

4. **Efficiency:** Enables quick access to files, directories, and devices.
5. **Multi-user Support:** Allows multiple users to work simultaneously without conflicts.

OR Q.5 (a) What is Access Control List?

Access Control List (ACL)

An **Access Control List (ACL)** is a **security mechanism** used in operating systems and file systems to **specify which users or system processes are granted access to specific resources** (such as files, directories, or devices) and what operations they are allowed to perform. It provides **fine-grained control** over permissions beyond the standard read, write, and execute permissions.

How ACL Works

1. Each file or resource has an **associated ACL**.
2. The ACL contains a **list of entries**, each specifying:
    - The **user or group**
    - The **type of access allowed** (read, write, execute, delete, etc.)
3. When a user or process attempts to access the resource, the operating system **checks the ACL** to see if the requested operation is permitted.
4. If the entry allows the operation, access is granted; otherwise, it is denied.

Advantages of ACL

- Provides **fine-grained access control** beyond the simple owner/group/others model.
- Can specify **different permissions for multiple users and groups** on a single file or directory.
- Supports **enhanced security policies** in multi-user and networked environments.

Example of ACL

Suppose a file `report.txt` has the following ACL entries:

User/Group	Permission
Alice	Read, Write
Bob	Read
Admins	Read, Write, Execute

- Alice can **read and edit** the file.
- Bob can only **read** the file.
- Admins can **read, write, and execute** the file.

(b) Explain the concept of virtual machines and pure virtualization.

Sure! Here's a **detailed theoretical explanation** of **Virtual Machines (VMs) and Pure Virtualization**:

---

1. Virtual Machines (VMs)

Definition:
A **Virtual Machine (VM)** is a **software-based emulation of a physical computer** that runs an operating system and applications just like a real computer. It provides an **isolated environment** where multiple VMs can run on a single physical machine using virtualization technology.

Key Concepts of VMs

1. **Isolation:** Each VM is independent; errors or crashes in one VM do not affect others.
2. **Encapsulation:** A VM can be treated as a single file or set of files, making it easy to move, copy, or back up.
3. **Hardware Independence:** VMs are **abstracted from physical hardware**, allowing the same VM to run on different physical machines.
4. **Resource Sharing:** Multiple VMs share the physical machine's CPU, memory, storage, and network resources through a **hypervisor**.

Benefits of VMs

- Efficient utilization of hardware resources.
- Easy testing and development of software in isolated environments.
- Support for running multiple operating systems on the same hardware.
- Improved security and fault tolerance due to isolation.

---

2. Pure Virtualization

Definition:
Pure virtualization is a virtualization technique where the **virtual machine provides a complete simulation of the underlying hardware**, allowing an unmodified guest operating system to run directly on the VM without any changes.

How Pure Virtualization Works

1. A **hypervisor** (Virtual Machine Monitor) sits between the physical hardware and the guest OS.
2. The hypervisor **intercepts all privileged instructions** from the guest OS and emulates hardware behavior.
3. The guest OS operates as if it has **full control over hardware**, even though the hypervisor manages access to physical resources.

Characteristics of Pure Virtualization

- Guest OS does not need to be modified.
- Complete isolation between VMs.
- Can run any OS that is compatible with the virtual hardware.
- Provides near-complete hardware emulation.

Advantages of Pure Virtualization

- **High compatibility:** Works with unmodified operating systems.
- **Strong isolation:** VMs cannot interfere with each other.
- **Ease of management:** Snapshots, cloning, and migration of VMs are simple.

Disadvantages

- Slight performance overhead due to hardware emulation.
- Requires a capable hypervisor to manage resources efficiently.

(c) What is RAID? Explain various levels of RAID and it importance.

RAID (Redundant Array of Independent/Inexpensive Disks) and Its Levels

Definition:
RAID, which stands for **Redundant Array of Independent or Inexpensive Disks**, is a storage technology that allows multiple physical hard drives to work together as a single logical unit. The main goal of RAID is to improve **performance, reliability, and fault tolerance** of storage systems. By distributing data across multiple disks and maintaining

redundancy through techniques like mirroring and parity, RAID ensures that data is **safe even if one or more disks fail**. This technology is widely used in **servers, data centers, cloud storage, and mission-critical applications** where data integrity and availability are crucial.

Importance of RAID:
RAID plays a very important role in modern computing systems because it provides a combination of **speed, reliability, and cost-effectiveness**.

By allowing multiple disks to work together, RAID improves read and write performance since data can be accessed or stored on multiple disks simultaneously.

At the same time, RAID protects against data loss by using redundancy mechanisms such as mirrored copies or parity information, allowing the system to reconstruct lost data if a disk fails.

It also allows systems to **scale easily**, as additional disks can be added to increase storage capacity or performance without disrupting operations. Overall, RAID ensures **continuous availability of data**, which is essential for enterprises, cloud services, and other high-demand systems.

RAID Levels

1. RAID 0 (Striping)

- **Definition:** In RAID 0, the data is **split into small blocks and distributed across all the disks in the array**. Each disk holds only part of the data, and there is **no duplication or parity**.
- **Key Features:**
  - Provides **high read and write performance** because multiple disks can be accessed simultaneously.
  - Uses **all disk space efficiently**, as there is no redundancy.
  - Minimum of **2 disks required**.
- **Advantages:**
  - Significantly increases **data transfer speed**, making it ideal for tasks that require fast access.
  - Full storage capacity is available since no space is used for redundancy.
- **Disadvantages:**
  - **No fault tolerance**; if one disk fails, all data in the array is lost.
  - Not suitable for storing critical data.
- **Use Case:** Temporary storage, gaming PCs, video editing systems, or any scenario where **speed is more important than data security**.

---

2. RAID 1 (Mirroring)

- **Definition:** RAID 1 stores an **exact copy of the data on two or more disks**. Every write operation is duplicated on all disks in the mirrored set.

- **Key Features:**
  - Provides **high reliability and fault tolerance**.
  - Minimum of **2 disks required**.
  - Read operations can be **faster**, as data can be read from any mirrored disk.
- **Advantages:**
  - Can survive the **failure of one disk** without data loss.
  - Simple to implement and manage.
  - Improves **read performance** in certain situations.
- **Disadvantages:**
  - Only **50% of disk space** is usable because half is used for mirroring.
  - Write speed can be slightly slower due to data duplication.
- **Use Case:** Critical systems like **banking servers, medical records storage, and essential business data**, where **data safety is more important than storage efficiency**.

---

3. RAID 5 (Striping with Parity)

- **Definition:** RAID 5 distributes data **and parity information** across at least **three disks**. Parity is a calculated value used to **reconstruct data if a single disk fails**.
- **Key Features:**
  - Balances **performance, reliability, and storage efficiency**.
  - Provides **fault tolerance for one disk failure**.
  - Minimum of **3 disks required**.
- **Advantages:**
  - Can survive a **single disk failure** without data loss.
  - Offers **efficient use of storage**; only a portion is used for parity.
  - Read operations are fast since data is striped across multiple disks.
- **Disadvantages:**
  - Write operations are slower due to **parity calculation**.
  - Rebuilding data after a disk failure can be **time-consuming** and affect performance.
- **Use Case:** Widely used in **file servers, database servers, and enterprise storage systems** where a **balance of reliability and performance** is required.

---

4. RAID 6 (Striping with Double Parity)

- **Definition:** RAID 6 is an extension of RAID 5 that uses **double parity**, allowing the array to survive **two simultaneous disk failures**. Data and two sets of parity information are striped across all disks.
- **Key Features:**
  - Provides **higher fault tolerance** than RAID 5.
  - Requires **minimum 4 disks**.
  - Suitable for systems that need **high availability and critical data protection**.
- **Advantages:**
  - Can tolerate the **failure of two disks simultaneously**.
  - Provides **high reliability** for large storage arrays.

- **Disadvantages:**
  - Write operations are slower due to **extra parity calculations**.
  - Slightly lower storage efficiency than RAID 5.
- **Use Case:** Ideal for **cloud storage systems, large data centers, and enterprise servers** that need **maximum fault tolerance**.

---

5. RAID 10 (1+0, Mirrored Stripes)

- **Definition:** RAID 10 combines **mirroring (RAID 1) and striping (RAID 0)**. Data is first mirrored, and then the mirrored pairs are striped across multiple disks.
- **Key Features:**
  - Provides **both high performance and high fault tolerance**.
  - Requires a **minimum of 4 disks**.
  - Can tolerate multiple disk failures as long as **no mirrored pair is completely lost**.
- **Advantages:**
  - Very high read and write speeds due to striping.
  - Can survive multiple disk failures if they are in different mirrored sets.
  - Ideal for **high-performance and critical applications**.
- **Disadvantages:**
  - Storage efficiency is only 50%, as half of the disk capacity is used for mirroring.
  - More expensive due to the need for **double the number of disks**.
- **Use Case:** High-performance databases, transactional systems, and **mission-critical applications** where **speed and data safety are both essential**.
